Relatório Final – Iniciação Científica

Estudo e Aplicação de Modelos de Previsão no Contexto de Séries de Vazões de Rios

Submetido à
Pró-Reitoria de Pesquisa da Unicamp

Departamento de Engenharia de Computação e Automação Industrial (DCA)
Faculdade de Engenharia Elétrica e de Computação (FEEC)
Universidade Estadual de Campinas (UNICAMP)
CEP 13083-852, Campinas, São Paulo (SP)

Candidato: Daniel da Costa Nunes Resende Neto
Orientador: Prof. Levy Boccato

1 Introdução

Na primeira etapa deste projeto, foram estudados dois modelos de predição clássicos da literatura, ambos de natureza linear: auto-regressivo (AR) e auto-regressivo de médias móveis (ARMA) [1], detalhados nas Seções 2.1 e 2.3 do relatório parcial.

Ambos foram implementados, com base em elementos da metodologia Box-Jenkins [1], e estudados no contexto da série de vazões de Água Vermelha. No caso do modelo ARMA, exploramos uma meta-heurística denominada *clearing* [2, 3] para o ajuste de seus parâmetros, visando minimizar o erro quadrático médio na etapa de treinamento (vide Seção 3.2 do relatório parcial).

Por fim, ambos os modelos foram comparados a partir do Erro Quadrático Médio (EQM) gerado para os dados de teste da série de Água Vermelha. Os desempenhos atingidos foram satisfatórios e relativamente próximos: enquanto o AR alcançou um EQM de 1088,6 $(\mathrm{m}^3/\mathrm{s})^2$, o ARMA obteve um EQM de 1056,3 $(\mathrm{m}^3/\mathrm{s})^2$, conforme apresentado na Seção 4 do relatório parcial.

Na segunda parte do projeto, os modelos AR e ARMA foram aplicados à série de vazões de Jirau (ver Seção 3.5), a qual apresenta um perfil hidrológico bastante diferente em relação à de Água Vermelha. Também, incorpora-se o estudo (ver Seção 2.1) e a implementação (ver Seções 3.2 a 3.4) de métodos não-lineares baseados em redes neurais [4] para a previsão de ambas as séries. São eles: a rede *perceptron* de múltiplas camadas (MLP, do inglês *multilayer perceptron*), como representante da classe de modelos *feedforward*; assim como três modelos recorrentes: as RNNs (do inglês, *recurrent neural networks*) simples, as LSTMs (do inglês, *long short-term memory*) [5] e as GRUs (do inglês, *gated recurrent units*) [6].

Então, foi realizada uma análise comparativa entre estes seis modelos no contexto das duas séries de vazões escolhidas, tendo como base o EQM para o conjunto de teste. A metodologia completa, os resultados obtidos e as conclusões finais do projeto encontram-se nas Seções 3.6 e 4 deste documento.

2 Modelos de Previsão

2.1 Modelos não-lineares

Os modelos não-lineares estudados pertencem todos à classe de redes neurais artificiais (ANNs, do inglês *artificial neural networks*). Em síntese, as ANNs se inspiram em princípios ligados ao funcionamento do sistema nervoso, em particular, do cérebro,

sendo formadas por várias unidades simples de processamento, denominadas neurônios artificiais, que desempenham um papel análogo ao do neurônio biológico.

Um modelo tipicamente explorado em ANNs para o neurônio artificial [4] é mostrado na Figura 1.

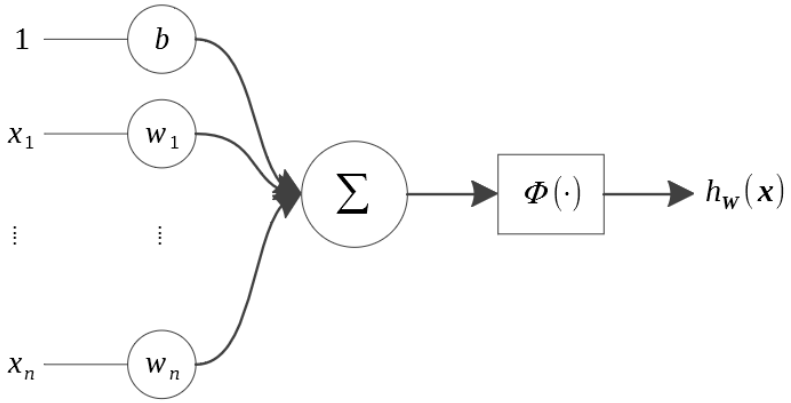

Figura 1: Modelo de neurônio artificial do tipo *perceptron*.

Neste modelo, os atributos x_i contidos na instância $\mathbf{x}$ e a saída $h_{\mathbf{w}}(\mathbf{x})$ são números reais, sendo que cada elemento de entrada x_i está associado a um peso $w_i \subset \mathbf{w}$. Aplica-se, então, uma função de ativação ϕ (discutida em breve) sobre a soma ponderada de suas entradas mais um termo de *bias* (b) de forma a gerar a saída:

$$h_{\mathbf{w}}(\mathbf{x}) = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

A função de ativação ϕ pode assumir várias formas, sendo as mais utilizadas as funções: logística (também chamada, em inglês, *sigmoid*), tangente hiperbólica (Tanh), ReLu e SeLu.

Em algumas arquiteturas de redes recorrentes, o modelo de neurônio pode apresentar algumas diferenças importantes. Desde modo, deixaremos sua descrição para a Seção 2.1.2, a qual introduz os detalhes deste tipo de rede.

2.1.1 MLP

As MLPs são redes neurais formadas fundalmentamente por múltiplas camadas de neurônios do tipo *perceptron*. A Figura 2 apresenta a estrutura geral de uma MLP.

Esta rede é composta, primeiramente, por uma camada ($l = 0$) de entrada. Os neurônios desta camada apenas passam para suas saídas o atributo com o qual foram alimentados.

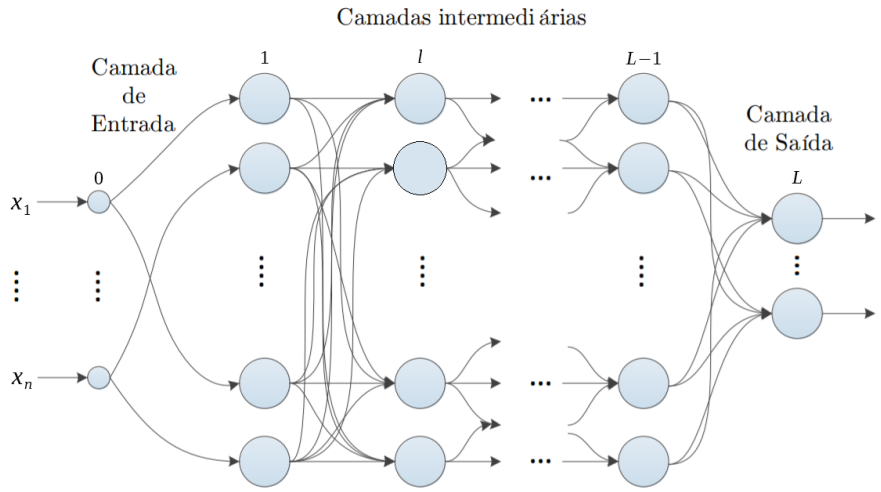

Figura 2: Arquitetura de uma MLP. Figura extraída de [7].

As próximas camadas, exceto a última, são denominadas camadas intermediárias (em inglês, *hidden layers*). Em cada camada, os estímulos de entrada são combinados linearmente, com base nos pesos de cada neurônio, e a função de ativação é aplicada sobre o resultado acumulado. Assim, as saídas dos neurônios de uma camada são propagadas adiante para a próxima camada, até que cheguem à última camada da rede.

Uma vez que todos os neurônios da camada l estejam conectados a todos os neurônios da próxima camada $(l+1)$, diz-se que estas camadas são totalmente conectadas (em inglês, *fully connected*), ou densas (em inglês, *dense*).

Por fim, a camada de saída (em inglês, *output layer*) é quem gera as respostas da rede para uma instância.

A equação (1) pode ser generalizada matricialmente (ver equações em (2)) para definir a matriz contendo os vetores de saídas de uma camada gerados a partir de um subconjunto de instâncias chamado *mini-batch*. Isto é feito separadamente para a camada de entrada (equação (2a)), uma camada intermediária (equação (2b)) e a camada de saída (equação (2c)), da seguinte forma:

$$\mathbf{H}^0(\mathbf{X}) = \mathbf{X}, \tag{2a}$$

$$\mathbf{H}^l_{\mathbf{W^l},\mathbf{B^l}}(\mathbf{H}^{l-1}) = \phi(\mathbf{H}^{l-1}\mathbf{W}^l + \mathbf{B}^l), \tag{2b}$$

$$\mathbf{H}^L_{\mathbf{W^L},\mathbf{B^L}}(\mathbf{H}^{L-1}) = \phi(\mathbf{H}^{L-1}\mathbf{W}^L + \mathbf{B}^L) \tag{2c}$$

Nestas equações:

- A matriz $\mathbf{X} = [\mathbf{x}_1\ \mathbf{x}_2\ \ldots\ \mathbf{x}_d]^\top$, com $\mathbf{X} \in \mathbb{R}^{d \times n}$, é o *mini-batch* recebido pela rede e contém d instâncias $\mathbf{x}_i^\top$ em suas linhas, sendo que cada instância é caracterizada por n atributos;

- Os vetores coluna $\mathbf{w}_j^l$ da matriz $\mathbf{W}^l = [\mathbf{w}_1^l\ \mathbf{w}_2^l\ \ldots\ \mathbf{w}_{m(l)}^l]$, com $\mathbf{W}^l \in \mathbb{R}^{m(l-1) \times m(l)}$, contêm os conjuntos de pesos pertencentes a cada um dos $m(l)$ neurônios da camada l. Note que o número de pesos de um neurônio da camada l (a dimensão dos vetores $\mathbf{w}_j^l$) é igual ao número de neurônios da camada anterior $m(l-1)$, já que este é também seu número de saídas. Os valores assumidos por $m(l)$ quando $l = 1, ..., L$ são definidos na construção da rede e, como as saídas da camada de entrada são as próprias entradas, $m(0) = n$;

- A matriz $\mathbf{H}^l = [\mathbf{y}_1^l\ \mathbf{y}_2^l\ \ldots\ \mathbf{y}_d^l]^\top$, com $\mathbf{H}^l \in \mathbb{R}^{d \times m(l)}$, contém os d vetores linha de saídas $\mathbf{y}_i^{l\top}$ da camada l. Cada um deles contém $m(l)$ saídas;

- Os vetores linha $\mathbf{b}_i^{l\top}$ da matriz $\mathbf{B}^l = [\mathbf{b}_1^l\ \mathbf{b}_2^l\ \ldots\ \mathbf{b}_d^l]^\top$, com $\mathbf{B}^l \in \mathbb{R}^{d \times m(l)}$, são idênticos entre si e contêm os $m(l)$ valores de *bias* de cada neurônio da camada l;

- A função de ativação ϕ é aplicada a todos os elementos da matriz $(\mathbf{H}^{l-1}\mathbf{W}^l + \mathbf{B}^l)_{d \times m(l)}$, a qual contém em seu elemento (i, j) a soma dos elementos do i-ésimo (de um total de d) vetor de saída da camada anterior ponderados pelos pesos do j-ésimo neurônio da camada l, mais o termo de bias.

O objetivo agora é treinar a rede. Em outras palavras, encontrar os valores de todos os pesos sinápticos, $\mathbf{W}^l, l = 1, \ldots, L$, que minimizem uma medida de erro entre as saídas geradas pela rede e as saídas desejadas.

No contexto de regressão/predição, um critério classicamente explorado é o EQM. O processo de treinamento dá origem, então, a um problema de otimização não-linear irrestrito. Neste contexto, o uso de algoritmos iterativos baseados em gradiente constitui a opção padrão na literatura, inclusive para modelos profundos [8]. Para isso, é preciso computar a derivada da função custo (EQM) com respeito a todos os pesos da rede, o que pode ser feito com o auxílio do algoritmo de retropropagação do erro (em inglês, *error backpropagation*) [9].

O otimizador clássico utilizado para a atualização dos pesos da rede é o gradiente descendente [4]. Contudo, a otimização por gradiente descendente sofre comumente com o desaparecimento dos gradientes (em inglês, *vanishing gradients problem*), ou ainda o contrário: o seu aumento desenfreado (em inglês, *exploding gradients problem*), que é

mais comum no contexto de redes recorrentes. Isso acontece à medida que os gradientes de erro são propagados para as camadas inferiores da rede, especialmente se esta for densa.

Uma técnica desenvolvida para tentar evitar ambos os problemas supracitados é a normalização do *batch* (BN, do inglês *batch normalization*) [4]. Ela consiste na adição de uma operação dentro do modelo logo antes ou depois da função de ativação de cada camada intermediária.

Esta operação simplesmente centra todas as instâncias ao redor do zero e as normaliza, resultando na nova variável $\hat{\mathbf{x}}_i$. Para isso, o algoritmo deve estimar a média e o desvio padrão de cada atributo dentro do *mini-batch* (daí o nome) que alimenta a camada de *batch normalization*.

O uso de BN introduz dois novos parâmetros por camada: um que escala os atributos normalizados $\boldsymbol{\gamma}$ e outro que os desloca $\boldsymbol{\beta}$, de modo a formar a saída $\mathbf{z}_i = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}_i + \boldsymbol{\beta}$. Cabe ao modelo aprender a escala e o deslocamento ótimo de cada atributo para cada camada.

Outro problema característico do gradiente descendente é sua lentidão, a qual pode ser contornada com a utilização de outros otimizadores, os quais também dependem essencialmente das derivadas parciais de primeira ordem (Jacobianos), mas incorporam estratégias que visam acelerar a convergência, seja por um refinamento na direção de ajuste, seja pelo controle do passo. O otimizador explorado neste trabalho é o Nadam (do inglês, *Nesterov adaptive moment estimation*) [4].

No método clássico do gradiente descendente, o peso $\mathbf{w}$ é atualizado diretamente subtraindo-se o gradiente da função custo relativo a ele naquele instante. Já a otimização por momento presente no Nadam incorpora a influência de gradientes passados, não apenas o local. Isto é feito subtraindo-se o gradiente local, desta vez, do vetor de momento $\mathbf{m}$, o qual é efetivamente empregado no ajuste do vetor $\mathbf{w}$.

O gradiente passa, então, a transmitir a ideia de aceleração, e não mais velocidade. Assim, o algoritmo acumula velocidade de convergência caso esteja caminhando para um ótimo local e escapa de platôs mais rapidamente.

Para que o algoritmo não avance velozmente em uma direção que não aponta para o ótimo local (como acontece no metódo clássico do gradiente descendente), introduz-se o vetor $\mathbf{s}$, o qual escala o vetor momento.

Esta técnica causa, no geral, uma diminuição no gradiente, especialmente se este toma grandes proporções. Assim, é possível corrigir a sua direção para um ponto mais próximo do ótimo local antes de uma convergência acelerada do algoritmo.

Por fim, o Nadam também incorpora a ideia proposta pelo matemático Yurii Nesterov (daí o nome) de que o gradiente deve ser medido não sobre o ponto atual $\mathbf{w}$, mas sobre

um ponto levemente à frente na direção do vetor momento. Esta ideia pode ser benéfica, uma vez que a direção deste aponta geralmente na direção correta (a direção do ponto ótimo), tornando a atualização dos pesos mais precisa.

Com isso, concluímos o estudo teórico da rede *feedforward* MLP e dos elementos básicos de uma rede neural (como a função de ativação e o otimizador), os quais serviram de base para este projeto. Nosso foco se direcionará, nas próximas seções, às redes neurais do tipo recorrente.

2.1.2 RNN

Uma RNN apresenta uma estrutura bastante similar à de uma MLP, mas com a presença de retroalimentações. A célula básica de uma RNN consiste em um único neurônio recorrente, o qual recebe entradas (mais um termo *bias*) e produz uma saída. Esta é alimentada à sua própria entrada.

A cada instante de tempo t (também chamado de *frame*), este neurônio recebe o vetor de entradas $\mathbf{x}_{(t)}$ assim como sua própria saída do instante de tempo anterior, $y_{(t-1)}$. Como não há saída prévia no instante 0, o primeiro valor de y é nulo.

A criação de uma camada de neurônios recorrentes é intuitiva. Cada um dos neurônios continua recebendo o vetor de entradas $\mathbf{x}_{(t)}$, porém passa a receber também um vetor de saídas $\mathbf{y}_{(t-1)}$, o qual contém as saídas passadas dos neurônios desta camada. Esta estrutura pode ser vista na Figura 3 (esquerda).

Para facilitar o entendimento, representamos esta rede através do tempo, como mostra a Figura 3 (direita). Isto é chamado de desenrolamento da rede no tempo, pois o mesmo neurônio é representado várias vezes, conforme o número de instantes de tempo considerados.

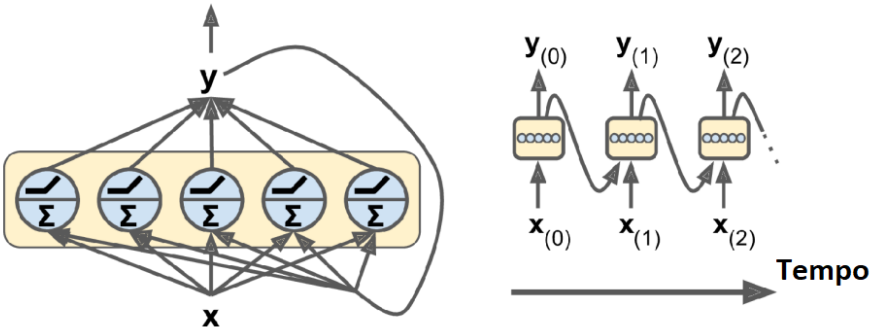

Figura 3: Camada de neurônios recorrentes. Figura extraída de [4].

É necessário, então, que cada neurônio recorrente possua dois vetores de pesos: o primeiro, $\mathbf{w}_x$, pondera as entradas instantâneas $\mathbf{x}_{(t)}$; o segundo, $\mathbf{w}_y$, é responsável por

ponderar as saídas passadas, $\mathbf{y}_{(t-1)}$. Considerando uma camada recorrente, podemos agrupar todos os pesos em duas matrizes, $\mathbf{W}_x$ e $\mathbf{W}_y$, de maneira que o vetor de saída da camada pode ser escrito como:

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^{\top}\mathbf{W}_x + \mathbf{y}_{(t-1)}^{\top}\mathbf{W}_y + \mathbf{b}^{\top}\right)^{\top} \tag{3}$$

sendo $\mathbf{b}$ o vetor de *bias*.

De modo análogo ao que foi feito para a MLP, podemos agrupar várias instâncias em um único *mini-batch* $\mathbf{X}_{(t)}$, de modo que a matriz de saídas é dada por:

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{B}\right) \\ &= \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)}\end{bmatrix}\mathbf{W} + \mathbf{B}\right) \text{ com } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}\end{aligned} \tag{4}$$

onde a matriz $\mathbf{Y}_{(t)}$ contém em cada linha as saídas de uma instância do *mini-batch*; a matriz $\mathbf{X}_{(t)}$ contém em cada linha uma diferente instância de tamanho igual à quantidade de atributos; as matrizes $\mathbf{W}_x$ e $\mathbf{W}_y$ contêm em cada coluna os pesos que multiplicam os atributos e as saídas prévias de uma instância, respectivamente; a matriz $\mathbf{B}$ contém o mesmo vetor de *bias* de cada neurônio em cada uma de suas linhas; e as matrizes $\mathbf{W}_x$ e $\mathbf{W}_y$ podem ser concatenadas em uma única matriz $\mathbf{W}$, como mostrado.

Devido à relação de recorrência presente em (4), $\mathbf{Y}_{(t)}$ acaba sendo influenciado por todas as entradas desde $t = 0$, *i.e.*, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(t)}$. Por isso, diz-se que a rede recorrente possui memória. A parte da rede que preserva estados ao longo do tempo é chamada de célula de memória (ou simplesmente célula).

A estrutura aqui apresentada (uma RNN simples) é uma célula muito básica, capaz apenas de aprender padrões curtos. As LSTMs e GRUs, apresentadas nas seções 2.1.3 e 2.1.4, buscam justamente a expansão desta memória.

Estas estruturas também realizam a retroalimentação a partir das próprias saídas y de cada neurônio. Contudo, é comum que se utilize uma função $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$ para representar o estado da célula atual e, consequentemente, para a retroalimentação. Já sua saída no instante t seria dada por $\mathbf{y}_{(t)} = g(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$, também uma função da entrada atual e do estado anterior.

Baseando-se nesta representação, uma RNN pode ser alimentada com uma sequência de entradas (isto é, um vetor de entradas por instante) e gerar uma sequência de saídas, como mostra o esquema no canto superior-esquerdo da Figura 4. Este tipo de rede é chamado sequência-sequência.

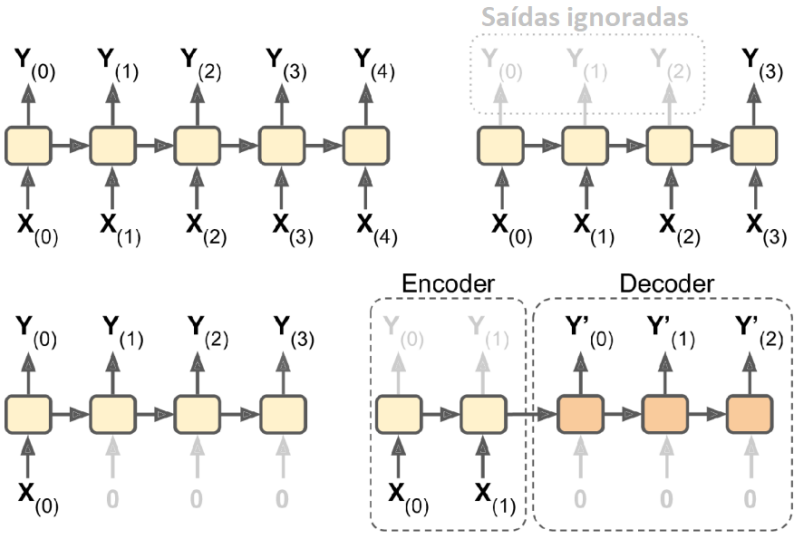

Figura 4: Modos de implementação de uma RNN. Figura extraída de [4].

Por outro lado, a rede pode ser alimentada com uma sequência de entradas e produzir uma única saída de interesse no último instante de tempo, como mostra o diagrama no canto superior-direito da Figura 4. Esta é uma rede sequência-vetor.

No contexto de séries temporais, ambos estes tipos de rede conseguem prever os $N-1$ valores de uma série de tamanho N mais os m próximos. Isto foi feito na segunda parte deste trabalho e está detalhado na Seção 3.2.

Por fim, a rede pode ser do tipo vetor-sequência, como mostra o esquema no canto inferior-esquerdo da Figura 4. A aplicação mais comum neste caso é a geração automática de rótulos para imagens.

É possível também utilizar um modelo sequência-vetor seguido por um vetor-sequência, chamado *Encoder-Decoder*. Ele é muito utilizado, por exemplo, para traduzir sentenças. Este último processo está representado no canto inferior-direito da Figura 4.

Uma RNN é treinada utilizando-se a técnica de retropropagação do erro através do tempo (do inglês, *backpropagation through time*). Para este fim, aplica-se a retropropagação do erro (analogamente ao que foi feito para a MLP) à RNN representada através de n instantes de tempo (veja a Figura 5).

Assim como na retropropagação regular, o primeiro passo consiste em gerar todas as saídas $\mathbf{Y}$ de todos os instantes de tempo (*forward pass*). Esse é representado pelas setas em pontilhado na Figura 5.

A função de perda E é, então, aplicada sobre todas as saídas relevantes (no caso da Figura 5, as três últimas). Os gradientes desta função em relação aos pesos das matrizes $\mathbf{W}$ e $\mathbf{B}$ são calculados e propagados em direção aos menores instantes de tempo (representado

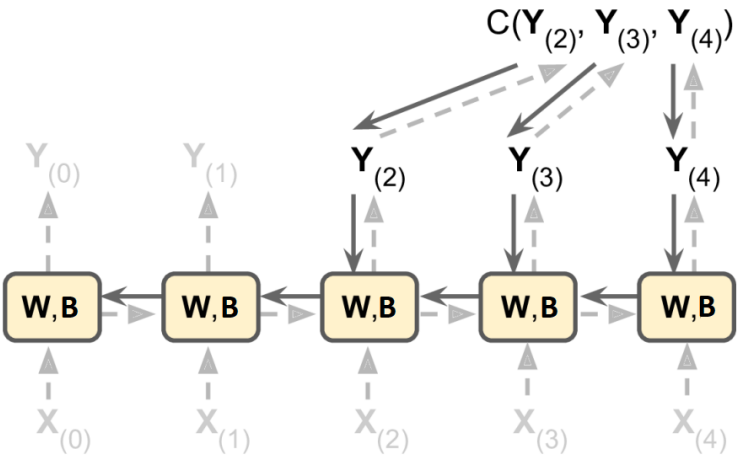

Figura 5: Retropropagação do erro através do tempo. Figura extraída de [4].

pelas setas sólidas). Contudo, esta propagação só acontece para os instantes de tempo cuja saída é considerada relevante (no exemplo, os instantes 2, 3 e 4).

É importante destacar que, na maior parte dos casos, utilizam-se funções de ativação como a tangente hiperbólica (*Tanh*) ou a logística (*sigmoid*) para redes recorrentes. Estas funções saturam dada uma entrada muito grande ou muito pequena. Assim, é possível contra-atacar um possível problema com o desaparecimento ou crescimento exacerbado dos gradientes.

2.1.3 LSTM

Como mencionado anteriormente, um dos maiores problemas com a RNN simples é sua memória. Devido às transformações que o dado de entrada sofre ao longo dos instantes de tempo, uma parte da informação é perdida. Eventualmente, não sobra nenhum traço das primeiras entradas.

Para abordar esse problema, diversos tipos de células com memórias a longo-prazo foram criadas. A mais popular entre elas é a LSTM [5].

Vista de fora, uma célula LSTM pode ser utilizada analogamente a um neurônio recorrente simples. A diferença é que seu estado é dividido em dois vetores diferentes: o vetor de longo prazo $\mathbf{c}$ e o vetor de curto-prazo $\mathbf{h}$. A estrutura interna de uma célula LSTM está representada na Figura 6.

A ideia principal é que a rede consiga aprender o que deve ser guardado no estado a longo prazo $\mathbf{c}_{(t)}$, e daquilo que foi guardado, o que deve ser descartado e o que deve ser considerado.

A primeira alteração que ocorre no vetor $\mathbf{c}_{(t-1)}$ é a perda de parte de sua memória,

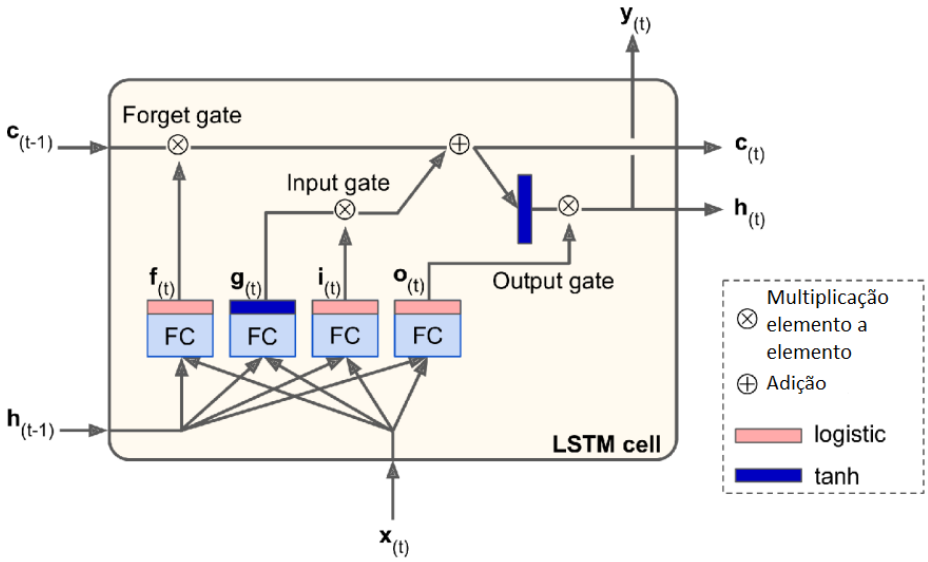

Figura 6: Estrutura interna de uma célula LSTM. Figura extraída de [4].

através da multiplicação elemento por elemento no que é chamado porta do esquecimento (do inglês, *forget gate*).

Depois, são incorporadas ao vetor novas memórias, selecionadas pela porta de entrada (do inglês, *input gate*) através da operação de adição. O resultado $\mathbf{c}_{(t)}$ é mandado então para a saída da célula. Porém, após a operação de adição, uma cópia do vetor $\mathbf{c}_{(t)}$ passa pela função tangente hiperbólica e é filtrado (novamente, com uma multiplicação termo a termo) pela porta de saída (do inglês, *output gate*). O resultado constitui o estado de curto prazo $\mathbf{h}_{(t)}$ o qual é também a saída da célula naquele instante $\mathbf{y}_{(t)}$.

Agora, olharemos para a entrada da LSTM. Tanto o vetor de entrada $\mathbf{x}_{(t)}$ quanto o vetor de estado de curto prazo do instante passado, $\mathbf{h}_{(t-1)}$, são fornecidos a quatro camadas diferentes: $\mathbf{f}_{(t)}$, $\mathbf{g}_{(t)}$, $\mathbf{i}_{(t)}$ e $\mathbf{o}_{(t)}$.

O vetor $\mathbf{g}_{(t)}$ é a saída da camada principal. Sua função é analisar a entrada $\mathbf{x}_{(t)}$ e o estado de curto-prazo prévio $\mathbf{h}_{(t-1)}$. As partes mais relevantes de $\mathbf{g}_{(t)}$ são guardadas no vetor $\mathbf{c}_{(t-1)}$, enquanto as partes menos relevantes são descartadas.

As camadas restantes são controladores, cujas saídas vão de 0 a 1 e alimentam as portas da LSTM. Todas as saídas participam de uma multiplicação elemento a elemento, de modo que um elemento igual a 0 fecha a porta e um elemento igual a 1 abre a porta.

Deste modo, o vetor $\mathbf{f}_{(t)}$ consegue controlar quais partes do vetor $\mathbf{c}_{(t-1)}$ devem ser apagadas. O vetor $\mathbf{i}_{(t)}$ consegue controlar quais partes de $\mathbf{g}_{(t)}$ devem ser adicionadas ao vetor $\mathbf{c}_{(t-1)}$. Finalmente, o vetor $\mathbf{o}_{(t)}$ consegue controlar quais partes de $\mathbf{c}_{(t-1)}$ devem formar tanto a saída $\mathbf{y}_{(t)}$ quanto o estado de curto-prazo $\mathbf{h}_{(t)}$.

Resumidamente, uma célula LSTM é capaz de detectar uma entrada relevante (pela

porta de entrada), guardá-la no vetor de estado a longo-prazo o quanto for necessário (pela porta do esquecimento) e extraí-la quando necessário (pela porta de saída). Isso traz à tona a capacidade elevada (em relação a uma célula recorrente simples) de reconhecimento e preservação de tendências presentes, no contexto deste projeto, em série temporais.

2.1.4 GRU

A célula GRU é uma versão simplificada da célula LSTM. Sua estrutura interna está representada na Figura 7.

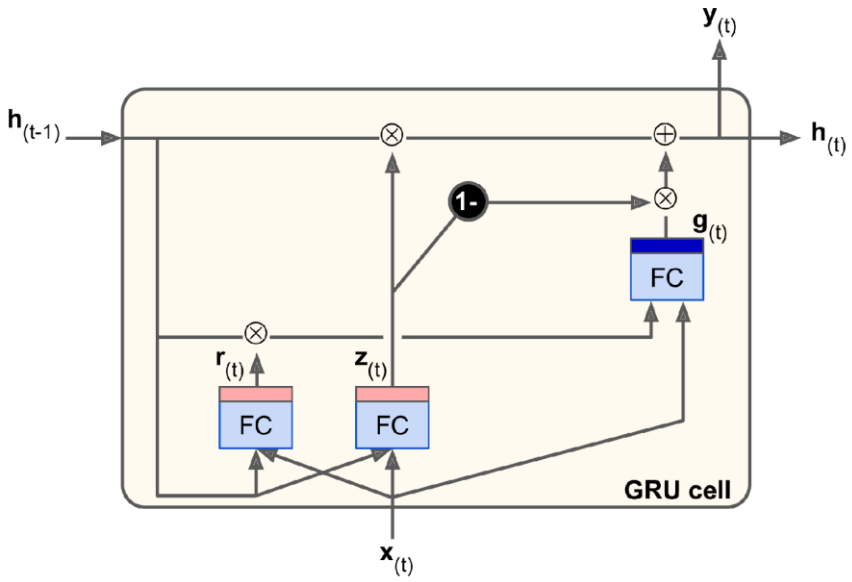

Figura 7: Estrutura interna de uma célula GRU. Figura extraída de [4].

Como pode ser notado, há agora somente um vetor de estado, $\mathbf{h}$. Também, um único vetor $\mathbf{z}_{(t)}$ controla tanto a porta de esquecimento (*forget gate*) quanto a porta de entrada (*input gate*). Se sua saída for igual a 1, a porta de esquecimento fica aberta e a de entrada fica fechada ($1 - 1 = 0$). Se sua saída for 0, o oposto acontece. Em outras palavras, para que uma memória seja mantida, o conteúdo do local em questão deve ser primeiramente apagado.

Além disso, não existe mais uma porta de saída (*output gate*). A todo instante, a saída da célula $\mathbf{y}_{(t)}$ e seu estado atual $\mathbf{h}_{(t)}$ são resultados diretos da operação de adição. Contudo, existe agora o vetor $\mathbf{r}_{(t)}$, o qual decide qual parte do vetor de estado prévio $\mathbf{h}_{(t-1)}$ será mostrado à camada principal (de saída $\mathbf{g}_{(t)}$). É importante notar que a capacidade de memória de uma LSTM ou uma GRU é, ainda assim, limitada, de modo que o aprendizado de padrões bastante afastados no tempo pode ser difícil.

Isto finaliza a exibição do embasamento teórico utilizado na segunda parte deste projeto. Nas próximas seções, apresentaremos características relevantes da série de Jirau,

bem como a divisão de dados aplicada para o modelo *feedforward* (MLP) e para os modelos recorrentes (RNN simples, LSTM e GRU). As etapas serão explicadas de modo genérico para, depois, serem detalhadas nas exposições das implementações particulares da MLP e das RNNs, referenciando sempre a teoria discutida na Seção 2. Concomitantemente, os resultados obtidos para ambos os modelos lineares e não-lineares na etapa de teste serão relatados, analisados e, ao final, comparados entre si.

3 Metodologia e Resultados

Nesta segunda parte do projeto, foram implementados os quatro tipos de redes neurais (MLP, RNN simples, LSTM e GRU) para cada uma das séries (Água Vermelha e Jirau), assim como o modelo AR e ARMA para a série de Jirau. Estas implementações serão abordadas nas próximas seções (3.1 a 3.4) assim como a descrição da série de Jirau. Ao final, os resultados (ver Seção 3.6) expostos cobrirão o projeto como um todo, e este será concluído na Seção 4.

3.1 Descrição da Série e Divisão de Dados

A série de vazões fluviais diárias (em m^3/s) utilizada nesta segunda parte do projeto foi a da usina de Jirau, considerando-se o mesmo período de 16 anos utilizado para a série de Água Vermelha (de 1º de janeiro de 2000 a 31 de dezembro de 2015).

Novamente, os primeiros 5114 dias (1º de janeiro de 2000 a 31 de dezembro de 2013) constituíram os dados de treinamento e validação, os quais foram fornecidos ao modelo de previsão. Já os últimos 730 dias (1º de janeiro de 2014 a 31 de dezembro de 2015) foram usados na etapa de teste.

As vazões da série de Jirau (de média 17 772 m^3/s) são significativamente maiores do que as de Água Vermelha (de média 2059,1 m^3/s), assim como seu desvio padrão e a diferença entre vazão máxima e mínima. Deste modo, foi possível trazer variedade às análises entre os métodos de previsão empregados neste projeto.

3.2 Implementação

A mesma metodologia foi empregada durante o projeto e avaliação da MLP e das redes neurais recorrentes (RNN simples, LSTM e GRU), de modo a tornar verossímil a comparação de seus respectivos desempenhos. A implementação destas redes foi dividida em duas etapas: validação e teste.

Contudo, há dois pontos a serem destacados em relação à divisão de dados da MLP e das redes recorrentes. O primeiro é que, enquanto a validação cruzada (do inglês, *cross validation*) do tipo *holdout* [10] foi utilizada para as redes recorrentes (assim como nos modelos AR e ARMA), uma validação cruzada do tipo 4-*fold* foi utilizada para a MLP.

No primeiro caso, pelo fato de o fator temporal ser intrínseco ao treinamento de redes recorrentes, desencourajou-se a divisão do tipo *k-fold*, pois esta quebra a sequência da série temporal. Assim, novamente os primeiros 14 anos foram divididos nos 3653 dados de treinamento (1º de janeiro de 2000 até 31 de dezembro de 2009) e nos 1461 dados de validação (1º de janeiro de 2010 até 31 de dezembro de 2013). Já na etapa de testes, foram utilizadas as medidas de vazão referentes ao período de 1º de janeiro de 2014 até 31 de dezembro de 2015 (totalizando 730 dias).

No segundo caso, a ausência do fator temporal durante o treinamento da MLP permitiu a utilização da validação-cruzada do tipo *k-fold*, na tentativa de deixar mais robusta a busca pela melhor configuração da rede durante a etapa de validação.

O segundo ponto a ser destacado trata da divisão dos dados dentro das matrizes de entrada $\mathbf{X}$ e de saída esperada $\hat{\mathbf{Y}}$ as quais devem ser introduzidas a uma rede neural durante um treinamento supervisionado.

A matriz $\mathbf{X}$ contém, em cada uma de suas linhas, uma parte (de tamanho M) diferente da série original. As partes entre si estão deslocadas de um dia, de modo que a primeira linha contenha as vazões dos dias 1 a M, a linha 2 contenha as vazões do dia 2 a $M+1$, e assim por diante. Já o vetor $\hat{\mathbf{Y}}$ contém, em suas linhas, a próxima vazão esperada (isto é, do próximo dia) em relação àquela linha, começando portanto com $M+1$, depois $M+2$, e assim por diante.

Cada linha da matriz $\mathbf{X}$ corresponde a uma instância, e M é o número de atributos (no contexto de séries temporais, o número de atrasos) contidos nas instâncias. Um conjunto de d linhas da matriz $\mathbf{X}$, portanto, define um *mini-batch*; e as d saídas esperadas deste *mini-batch* se encontram nas d linhas equivalentes do vetor $\hat{\mathbf{Y}}$.

As matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ para um mesmo número de atrasos M são as mesmas para a MLP e para as redes recorrentes. Porém, o modo como são interpretadas estas matrizes, no contexto de MLPs (Figura 2) e RNNs (Figura 5), é diferente.

No contexto de MLPs, as instâncias da matriz $\mathbf{X}$ são alimentadas, uma a uma, para todos os neurônios da primeira camada intermediária (observe a Figura 2). Já as linhas da matriz $\hat{\mathbf{Y}}$ são as saídas únicas da MLP, a qual possui um único neurônio na camada de saída.

Agora, no contexto de RNNs, as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ definem uma rede recorrente do

tipo sequência-vetor, a qual possui um único neurônio na camada de saída. O número de atrasos M equivale ao número de instantes de tempo da rede. Em outras palavras, cada um dos atributos de uma instância equivale à entrada única da rede em dado instante de tempo (observe a Figura 3), recebida por todos os neurônios da primeira camada intermediária.

Já as saídas esperadas em $\hat{\mathbf{Y}}$ equivalem ao elemento de saída da rede no último instante de tempo M considerado.

Esclarecidos os significados das divisões da série temporal nas matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ no contexto de MLPs e RNNs, cabe agora descrever as etapas de validação e teste aplicadas para estas redes neurais.

1. **Etapa de Validação**

    1.1. Escolhe-se um número de atrasos M pertencente a um intervalo I_M, e criam-se as duas matrizes $\mathbf{X}$ (uma para os dados de treinamento, e outra para os de validação) e as duas matrizes $\hat{\mathbf{Y}}$ correspondentes. No caso da MLP, o mesmo é feito, porém desta vez as matrizes $\hat{\mathbf{X}}$ e $\hat{\mathbf{Y}}$ são únicas, pois os vetores de treinamento e validação foram concatenados (em um de tamanho 3653) para esta etapa. O motivo por trás disto será explicado na Seção 3.3;

    1.2. Criam-se redes neurais (do tipo em questão) a partir da biblioteca com uma camada intermediária de $N \subset I_N$ neurônios (I_N variando com o tipo da rede) e uma camada de saída contendo um único neurônio. Treinam-se as redes construídas por um número de épocas, sendo seus desempenhos o menor EQM cometido para os dados de validação. Como a MLP possui 4 conjuntos distintos de validação (cada um correspondendo a 1/4 do vetor concatenado), o desempenho das redes criadas é dado pela média dos 4 (um para cada conjunto) EQMs finais de validação. Define-se, então, o número N de neurônios que obteve o melhor desempenho para aquele número de atrasos;

    1.3. O processo é repetido para outros valores de M pertencentes ao intervalo I_M. Ao final, consideramos os três melhores números de atrasos M e seus números de neurônios correspondentes.

2. **Etapa de Teste**

    2.1. As melhores configurações de cada rede (indicadas por uma combinação de número de atrasos e número de neurônios) são, então, utilizadas no retreinamento, feito desta vez com validação cruzada do tipo *holdout*. Não houve alteração na divisão de dados, isto é, os dados de validação e os de

treinamento correspondem aos mesmos 1461 e 3653 dias, respectivamente. No caso da MLP, o retreinamento é feito de duas maneiras: a primeira utilizando-se a validação cruzada do tipo *holdout* como descrito, e a segunda utilizando os vetores de treinamento e validação concatenados, sem validação;

2.2. Os modelos retreinados são usados para prever todas as vazões presentes nos dados de teste (de 1º de janeiro de 2014 a 31 de dezembro de 2015). Baseado nas saídas geradas, calcula-se o EQM cometido por cada conjunto;

2.3. O retreinamento é feito um total de 5 vezes para cada conjunto. A menor média dos EQMs para os dados de teste produzida entre os conjuntos é a medida do desempenho daquela rede na previsão da série em questão. A partir dela é feita a comparação de sua eficiência perante os outros modelos.

A descrição específica da implementação de cada modelo será detalhada nas Seções 3.3 e 3.4.

3.2.1 Motivação para a escolha dos atrasos M e K

Pelo mesmo motivo apresentado no relatório parcial, o número de atrasos M foi definido, a princípio, através da função de autocorrelação parcial aplicada sobre os dados de treinamento e validação concatenados da série de Jirau.

O intervalo escolhido começaria em 60 e terminaria em 110, porém, foram obtidos valores do menor EQM de validação maiores do que quando M tinha valor muito menor. Deste modo, optou-se, como no relatório parcial, pelo intervalo $I_M = [1, 30]$.

Já o intervalo de valores possíveis para K foi definido novamente a partir da função de autocorrelação para os dados de treinamento e validação referente à série de Jirau. Optamos pelo mesmo conjunto de atrasos do relatório parcial, $C_K = \{1, 5, 10, 20, 40\}$.

3.3 Implementação da MLP

Para a etapa de validação (ver Seção 3.2), deseja-se encontrar o valor de atraso M dentro de I_M (ver Seção 3.2.1) que leve ao menor EQM para os dados de validação.

Foi necessário apenas um laço externo que percorresse o intervalo I_M. Para dado número de colunas M, são construídas as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ a partir dos vetores de validação e treinamento concatenados (3653 dados).

A biblioteca Keras escrita em Python possui a função GridSearchCV, a qual realiza por completo a etapa de validação descrita na Seção 3.2. Como sugere seu nome, ela

aplica, por *default*, a validação cruzada do tipo *k-fold*. Assim, basta entregar o conjunto completo (no caso, o concatenado) das matrizes de entradas e saídas esperadas, pois a própria função divide-as nos 4 pares de treinamento e validação desejados.

Para que as diferentes redes com diferentes números N de neurônios sejam criadas, devem-se, antes, definir os parâmetros comuns (e invariáveis) entre elas. São tais parâmetros: a função de ativação, o otimizador, o tamanho do *batch* (em inglês, *batch size*), a inicialização de dados, a taxa de aprendizado (em inglês, *learning rate*) e a presença ou ausência do *batch normalization*.

Considerando todas as 4 redes neurais e os valores de $M \subset I_M$ no contexto da série de Água Vermelha, foi feita uma busca pela combinação ótima de parâmetros comuns, cujo resultado (o qual incorporou *batch normalization*) se encontra na Tabela 1.

Tabela 1: Parâmetros da MLP

Função de ativação	η	*Batch size*	Otimizador	Inicialização
Selu	0,003	32	Nadam	Lecun Normal

Definidos os parâmetros fixos das redes, a rotina GridSearchCV as treina por 150 épocas variando o número de neurônios dentro do conjunto $I_N^{\text{MLP}} = \{5, 10, 20, 30, 50, 80\}$.

Os resultados da etapa de validação para todos números de atrasos M equivalem ao menor EQM médio final para os dados de validação e seu número N de neurônios correspondente.

Os três melhores conjuntos $\{M, N\}$ de cada série foram retreinados cinco vezes, considerando tanto uma abordagem com validação cruzada do tipo *holdout*, quanto uma abordagem sem validação (na qual todos os dados de treinamento e validação servem para treinamento). Cada um deles produziu um EQM médio para os dados de teste. Os menores EQMs médios para cada série estão destacados na Tabela 2.

Tabela 2: Resultados Finais - MLP

Série	EQM $(m^3/s)^2$
Água Vermelha	1165 ± 79
Jirau	478170 ± 6554

3.4 Implementação das RNNs

O mesmo procedimento aqui descrito foi aplicado para as três redes recorrentes (RNN simples, LSTM e GRU), uma vez que, baseando-se na biblioteca Keras, basta substituir

o tipo de célula desejado para o treinamento de uma rede.

Novamente, para a etapa de validação (ver Seção 3.2), deseja-se encontrar o valor de atraso M dentro de I_M (ver Seção 3.2.1) que leve ao menor EQM para os dados de validação.

Foi necessário apenas um laço externo que percorresse o intervalo I_M. Para dado número de colunas M, são construídas as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ a partir dos dados de treinamento e validação separados, pois a validação desta vez é do tipo *holdout*.

Os parâmetros fixos utilizados para as redes recorrentes podem ser vistos na Tabela 3.

Tabela 3: Parâmetros das Redes Recorrentes

Função de ativação	η	*Batch size*	Otimizador	Inicialização
Tanh	0,01	32	Nadam	Glorot Uniform

Observe que, neste caso, não é feito o *batch normalization*, a taxa de aprendizado é de $0,01$ e a função de ativação utilizada é a tangente hiperbólica, assim como a inicialização é do tipo Glorot Uniform (estas duas últimas opções fazem parte do padrão (*default*) para redes recorrentes).

Como a função tangente hiperbólica apresenta saturação para entradas de módulos elevados, as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ foram escaladas para valores em torno de 0. Também decidiu-se realizar a diferenciação dos elementos da série antes que fossem definidas as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$. Esta técnica consiste em subtrair cada valor da série pelo seu elemento anterior, na tentativa de remover tendências ali presentes.

Após a diferenciação e, em seguida, o escalamento das vazões, as quatro matrizes são construídas e entregues à função GridSearchCV, duas como dados de treinamento e as outras como dados de validação.

Os conjuntos contendo os números N de neurônios considerados são:

1. $I_N^{\text{RNNsimples}} = I_N^{\text{GRU}} = \{5, 10, 15, 20, 30, 50, 75, 100\}$;
2. $I_N^{\text{LSTM}} = \{20, 30, 50, 75, 100, 150, 200, 300, 500\}$.

Encontrou-se o melhor número N de neurônios para cada atraso M, concluindo a etapa de validação.

Novamente, foram escolhidas para o retreinamento as três melhores combinações $\{M, N\}$ de cada rede recorrente para cada série. Os modelos foram retreinados cinco

vezes (desta vez apenas com validação do tipo *holdout*) e produziram EQMs médios para os dados de teste. Os menores EQMs médios para cada série e modelo estão destacados na Tabela 4.

Tabela 4: Resultados Finais - Redes Recorrentes

	Água Vermelha $(m^3/s)^2$	Jirau $(m^3/s)^2$
RNN simples	1048 ± 110	414914 ± 11516
LSTM	1322 ± 100	456127 ± 20792
GRU	1088 ± 98	502325 ± 40190

3.5 Resultados - Modelos Lineares

A fim de complementar a análise comparativa nesta segunda parte do projeto, repetimos o procedimento de construção e teste dos modelos AR e ARMA, detalhado no relatório parcial, considerando, agora, a série de Jirau. O desenvolvimento dos métodos lineares e todos os parâmetros relevantes são exatamente os mesmos que foram empregados no primeiro relatório.

O EQM e o EAM para os dados de teste do modelo AR são de: 393130 $(m^3/s)^2$ e 404 m^3/s, respectivamente. Já para o modelo ARMA, o EQM e o EAM para os dados de teste são: 614060 $(m^3/s)^2$ e 489 m^3/s, respectivamente.

3.6 Comparação entre modelos

Os EQMs para os dados de teste de cada modelo são colocados na Tabela 5 em ordem crescente (para a série de Água Vermelha).

Tabela 5: Resultados Finais

	Água Vermelha $(m^3/s)^2$	Jirau $(m^3/s)^2$
RNN simples	1048 ± 110	414914 ± 11516
ARMA	1056,3	614060
GRU	1088 ± 98	502325 ± 40190
AR	1088,6	393130
MLP	1165 ± 79	478170 ± 6554
LSTM	1322 ± 100	456127 ± 20792

O primeiro ponto a ser destacado é que o desempenho das células recorrentes foi superior ao desempenho da MLP. Consideremos uma MLP com uma única camada de

N_n neurônios. Esta rede contém, para instâncias de tamanho n, $n \times N_n$ pesos a serem ajustados. Já uma rede recorrente de camada única, e mesmo número de neurônios, necessita apenas ajustar N_n parâmetros (independentemente do tamanho da instância), pois estes são compartilhados entre os n instantes de tempo. O número reduzido de pesos da rede recorrente simplifica seu aprendizado.

Quanto ao acesso das redes à instância, de fato, a MLP a observa por completo, enquanto a rede recorrente acessa somente um de seus elementos em cada instante de tempo. A informação introduzida por cada um desses elementos é passada para os instantes de tempo subsequentes apenas pelo estado da célula, dificultando a assimilação de padrões longos. Contudo, como as instâncias apresentam um tamanho $n \in I_M$ reduzido, a célula de memória consegue aprender com êxito os padrões fundamentais presentes na instância. Isto minimiza a possível vantagem possuída pela MLP, de acesso à instância completa, e configura outro motivo pelo qual o desempenho da RNN simples foi superior.

Por outro lado, tanto a LSTM quanto a GRU não ofereceram melhorias em relação à RNN simples. Apesar de serem capazes de manter uma informação dentro da rede por um período de tempo mais longo, provavelmente nos cenários aqui tratados tal memória de longo prazo não era estritamente necessária. Assim, houve um gasto de recursos com uma arquitetura mais flexível que provou ser infrutífero para a otimização destas células. Deste modo, ambas as propostas (LSTM e GRU) acabaram sendo superadas pela RNN simples.

Os modelos lineares, por sua vez, se saíram muito bem. O estudo feito em [11] mostra que modelos estatísticos (como o AR e o ARMA) podem superar métodos de aprendizado de máquina (ML, em inglês *Machine Learning*), cuja sofisticação e demanda computacional são maiores. Isso mostra que um alto desempenho de extrapolação é acompanhado de grande eficiência. Em outras palavras, uma capacidade superior de predição pode ser atingida a partir de uma menor complexidade computacional (CC, em inglês *Computational Complexity*). O inverso também é verdadeiro, já que abordagens mais complexas de extrapolação por métodos de ML exibem resultados piores, indicando que robustez teórica nem sempre é favorável.

O estudo [11] também mostra que modelos que melhor ajustam os parâmetros da rede durante o treinamento não necessariamente apresentam maior capacidade de extrapolação. O exemplo dado pelo artigo é a LSTM, a qual "comparada com redes neurais mais simples, como a RNN e a MLP, demonstrou melhor *fitting*, porém pior precisão de predição [para os dados de teste]". Comparando-se, neste projeto, o desempenho de validação da LSTM ao desempenho de validação da RNN simples, nota-se esse mesmo comportamento: os EQMs

de validação da LSTM são, no geral, menores, porém a capacidade de extrapolação da RNN simples é superior no contexto de ambas as séries.

Em suma, as características das séries de Água Vermelha e Jirau foram bem representadas pelos modelos lineares AR e ARMA, uma vez que este obteve o segundo menor EQM final (1056,3 $(m^3/s)^2$) no contexto da série de Água Vermelha, e aquele obteve o menor EQM final (393130 $(m^3/s)^2$) no contexto da série de Jirau. Já a flexibilidade adicional dos modelos não-lineares acabou dificultando o aprendizado efetivo dos comportamentos das séries, com exceção da RNN simples, a qual apresentou um desempenho equiparável ao dos modelos lineares: o menor EQM final (1048 ± 110 $(m^3/s)^2$) no contexto da série de Água Vermelha e o segundo menor EQM final (414914 ± 11516 $(m^3/s)^2$) no contexto da série de Jirau.

4 Conclusões

Os avanços da inteligência artificial (AI, em inglês *Artificial Intelligence*) nos últimos anos permitiram a criação de veículos autônomos (AVs, em inglês *Autonomous Vehicles*), bem como de programas que dominam jogos (*e.g.*, xadrez, GO), que reconhecem e classificam expressões faciais, voz e escrita, que realizam traduções instantâneas, e que propõem diagnósticos médicos [12, 13]. Em muitas destas aplicações, o desempenho atingido é tido como superior ao do próprio ser humano.

É possível que métodos de ML aplicados à previsão de séries temporais alcancem um desempenho igualmente sublime e superior ao dos modelos lineares. Contudo, a implementação desses métodos é mais difícil, pois as regras para a predição de uma série são desconhecidas e mutáveis, existem instabilidades estruturais nos dados considerados e há aplicações em que as próprias predições podem alterar o futuro, aumentando ainda mais a incerteza das estimativas. Logo, pesquisas adicionais que experimentem ideias inovadoras e criem ajustes aos modelos de previsão são necessárias para ultrapassar a capacidade de extrapolação dos métodos estatísticos.

Neste contexto, introduzir o aluno a variadas estruturas de redes neurais, algumas com apelo moderno, o põe em contato com o cenário atual do problema de previsão de séries por aprendizado de máquina, o qual se encontra ainda em desenvolvimento e possui ampla possibilidade de expansão, seja na área da estatística, econometria, matemática financeira, ou mesmo ambiental. Além disso, este projeto fornece os meios de compreensão básicos das diversas aplicações de redes neurais (por exemplo aquelas citadas anteriormente) e dá a ele a oportunidade de atuar, um dia, em uma dessas áreas.

Referências

[1] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*, 5th ed. Wiley, 2015.

[2] L. Boccato, "Aplicação de computação natural ao problema de estimação de direção de chegada," Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 2010.

[3] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," *2nd Int. Conf. on Genetic Algorithms*, 1987.

[4] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. O'Reilly Media, Inc., 2019.

[5] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv:1412.3555 [cs.NE]*, 2014.

[7] L. Boccato, "Novas propostas e aplicações de redes neurais com estados de eco," Tese de doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 2013.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[9] S. Haykin, *Adaptive filter theory*, 5th ed. Prentice Hall, 2013.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning*. Cambridge, United Kingdom: Springer, 2011.

[11] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "Statistical and machine learning forecasting methods: Concerns and ways forward," *PLOS ONE*, vol. 13, no. 3, pp. 1–26, 03 2018. [Online]. Available: https://doi.org/10.1371/journal.pone.0194889

[12] H. Liang, B. Y. Tsui, and H. Ni, "Evaluation and accurate diagnoses of pediatric diseases using artificial intelligence," *Nat Med*, vol. 25, pp. 433–438, 2019. [Online]. Available: https://doi.org/10.1038/s41591-018-0335-9

[13] D. Ardila, A. P. Kiraly, and S. Bharadwaj, "End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography," *Nat Med*, vol. 25, pp. 954–961, 2019. [Online]. Available: https://doi.org/10.1038/s41591-019-0447-x